Data dependences and hazards

Determining how one instruction depends on another is **critical** to determining how much parallelism exist in a program and how that parallelism can be exploited. In particular, to exploit instruction-level parallelism, we must determine which instructions can be executed in parallel: if two instructions are parallel, they can be executed simultaneously in a pipeline of arbitrary depth without causing any stalls. If two instructions are dependent, instead, they are not parallel and must be executed in order.

There are three different types of dependences:

- Data dependences (also called *true* data dependences)
- Name dependences
- Control dependences

Since a dependence can limit the amount of instruction-level parallelism we can exploit, we must find ways to overcome these limitations. A dependence can be overcome in two different ways:

1. Maintaining the dependence but avoiding a hazard
2. Eliminating a dependence by transforming the program code

Scheduling code is the primary method used to avoid a hazard without altering a dependence, and such scheduling can be done both by the compiler and by the hardware.

Data dependences

An instruction j is data dependent on instruction i if either of the following holds:

- Instruction i produces a result that may be used by instruction j
- Instruction j is data dependent on instruction k, and instruction k is data dependent on instruction i

If we take a look at this MIPS code:

```
Loop:   L.D    F0, 0(R1)    ;F0=array element
                ADD.D  F4, F0, F2   ;add scalar in F2
```

```
S.D    F4, 0(R1)    ;store result
DADDUI R1, R1, #-8  ;decrement pointer by 8 bytes
BNE    R1, R2, LOOP ;branch R1 ≠ R2
```

We can find some data dependences:

- Between instructions 1 and 2, involving register `F0`
- Between instructions 2 and 3, involving register `F4`
- Between instructions 4 and 5, involving register `R1`

A data dependence conveys three things:

1. The possibility of a hazard
2. The order in which results **must** be calculated
3. An upper bound on how much parallelism can possibly be exploited

Name dependences

A name dependence occurs when two instructions use the same register or memory location, called a *name*, but there is no flow of data between the instructions associated with that name.

There are two types od name dependences between an instruction i that precedes an instruction j in program order:

1. An **antidependence** between instruction i and j occurs when instruction j writes a register or memory location that instruction i reads. The original value must be preserved to ensure that i reads the correct value.
2. An **output dependence** occurs when instruction i and instruction j write the same register or memory location. The ordering between instructions must be preserved to ensure that the value finally written corresponds to instruction j.

Both of these dependences are not true data dependences, since there is no value being transmitted between the instructions. Because of this, instructions involved in a name dependence can execute simultaneously or be reordered if the name used in the instructions is changed so the instructions do not conflict. This renaming can be more easily done for register operands, where it is called *register renaming*. Register renaming can be done either statically by a compiler or dynamically by the hardware.

Data hazards

A hazard exists whenever there is a name or data dependence between instructions, and they are close enough that the overlap during execution would change the order of access to the operand involved in the dependence.

Because of the dependence, we must preserve what is called *program order*: the order that the instructions would execute in if executed sequentially, one at a time, as determined by the original source program.

The goal of both our software and hardware techniques is to exploit parallelism by preserving program order **only where it affects the outcome of the program**.

Data hazards may be classified as one of three types, considering two instructions i and j with i preceding j:

1. **RAW** (read after write): j tries to read a source before i writes it, so j incorrectly gets the old value (true data dependence).
2. **WAW** (write after write): j tries to write an operand before it is written by i. The writes end up being performed in the wrong order, leaving the value written by i rather than the value written by j in the destination (output dependence).
3. **WAR** (write after read): j tries to write a destination before it is read by i, so i incorrectly gets the new value (antidependence).

Control dependences

The last type of dependence is a **control dependence**. A control dependence determines the ordering of an instruction i with respect to a branch instruction so that instruction i is executed in correct program order and only when it should be.

Every instruction, except for those in the first basic block of the program, is control dependent on some set of branches and, in general, these control dependences must be preserved to preserve the program order.

One of the simplest examples of a control dependence is the dependence of the statements in the “then” part of an if statement on the branch. For example, in the code segment:

```
if (p1) {
        S1;
}

if (p2) {
```

```
        S2;
}
```

`S1` is control dependent on `p1`, and `S2` is control dependent on `p2`, but not on `p1`.

In general, two constraints are imposed by control dependences:

1. An instruction that is control dependent on a branch cannot be moved before the branch so that its execution is no longer controlled by the branch. For example, we cannot take an instruction from the "then" portion of an if statement and move it before the if statement
2. An instruction that is not control dependent on a branch cannot be moved after the branch so that its execution is controlled by the branch. For example, we cannot take a statement before the if statement and move it into the then portion

When processors preserve strict program order, they ensure that control dependences are also preserved. We may be willing to execute instructions that should not have been executed, however, thereby violating the control dependences, if we can do so *without affecting the correctness of the program.* Thus, control dependence is not the critical property that must be preserved. Instead, the two properties critical to program correctness–and normally preserved by maintaining both data and control dependences–are the *exception behaviour* and *data flow.*

Preserving the exception behaviour means that any changes in the ordering of instruction execution must not change how exceptions are raised in the program.

A simple example is provided by this snippet of code:

```
    DADDU    R2, R3, R4
        BEQZ     R2, L1
        LW       R1, 0(R2)
L1:
```

In this case, if we don't maintain the data dependence involving `R2`, we can change the result of the program. Less obvious is the fact that if we ignore the control dependence and move the load instruction before the branch, the load instruction may cause a memory protection exception. Notice that *no data dependence* prevents us from interchanging the `BEQZ` and the `LW`; it is only the control dependence.

The second property preserved by maintenance of data dependences and control dependences is the data flow. The data flow is the actual flow of data values among instructions that produce results and those that consume them.

It is insufficient to just maintain data dependences because an instruction may be data dependent on more than one predecessor, given that branches make the data flow dynamic. Program order is what determines which predecessor will actually deliver a data value to an instruction, and it is ensured by maintaining control dependences.

For example, consider the following code:

```
        DADDU    R1, R2, R3
        BEQZ     R4, L
        DSUBU    R1, R5, R6
L:
        OR       R7, R1, R8
```

In this example, the value of `R1` used by the `OR` instruction depends on whether the branch is taken or not. Data dependence alone is not sufficient to preserve correctness. The `OR` instruction is data dependent on both the `DADDU` and `DSUBU` instructions, but preserving that order alone is insufficient for correct execution. Instead, when the instructions execute, the data flow must be preserved: if the branch is not taken, then the value of `R1` computed by the `DSUBU` should be used by the `OR`, and, if the branch is taken, the value of `R1` computed by `DADDU` should be used by the `OR`. By preserving the control dependence of the `OR` on the branch, we prevent an illegal change to the data flow.